Família de funções Apply

Vanderlei Júlio Debastiani (vanderleidebastiani@yahoo.com.br)

26 Maio 2020

Índice de conteúdo

1 Introdução

As funções da família *Apply* são alternativas aos loop/laços, elas pertencem ao pacote *base* do **R** e são projetadas para serem aplicadas a diferentes estruturas de dados como vetores, data.frames, matrizes e listas. As funções permitem cruzar os dados de várias maneiras e evitar o uso explícito de loop/laços.

Essas funções funcionam de maneira similar, todas requerem uma entradas de dados organizada em uma das estruturas básicas e aplicam uma segunda função a estrutura, sendo possível passar argumentos opcionais para a função aplicada. Os resultados da aplicação podem ser simples, como estátisticas descritivas e transformações ou mesmos retornar estruturas complexas como matrizes ou listas.

De uma maneira ampla, as funções desta família ajudam a executar operações com poucas linhas de código, e é composta principalmente por **apply**, **sapply**, **lapply**, **tapply** e **mapply**, e funções associadas que compartilham da mesma lógica como **sweep**, **replicate**, entre outras.

O tutorial requer conhecimento básico sobre **R**, neste os código são apenas brevemente comentados sendo que a ideia é que o leitor seja capaz de reproduzir os comandos aqui contidos, e busque mais informações sobre os argumentos extras nas páginas de ajuda de cada função. O texto está dividido em quatro partes: apresentação dos dados utilizados como exemplo, argumentos comuns, apresentação das principais funções da família *Apply* e teste de desempenho. Ainda, no final do texto, é apresentado um guia de ajuda rápida mostrando as principais funções apresentadas aqui.

2 Tipos de estruturas de dados

Para usar as funções da família *Apply* é preciso ter claro como são dispostas as estruturas de dados no **R**. As estruturas básicas são vetores, data.frames, matrizes e listas, com as seguintes características:

- **Vetores** (*vector*) - Conjuntos de dados unidimensionais do mesmo tipo;
- **Data.frames** (*data.frame*) - Conjuntos bidimensionais de vetores de mesmo comprimento, sendo que cada vetor pode ser de um tipo diferente. Nos data.frames os vetores são agrupados pelas colunas;
- **Matrizes** (*matrix*) - Conjuntos bidimensionais de dados do mesmo tipo. As matrizes não são consideradas agrupamento de vetores;
- **Listas** (*list*) - Conjuntos de dados de qualquer tipo, incluindo listas de listas. Cada elemento da lista pode ser considerado um vetor.

2.1 Dados de exemplo

Neste tutorial são usados dados simplificados, geradas apenas para exemplificar as estruturas de dados. Os códigos para gerar os dados e o arquivo *.RData* podem ser baixados em github.com/vanderleidebastiani/tutoriais.

```
# Carregar dados
urlRemote  <- "https://raw.githubusercontent.com/"
pathGithub <- "vanderleidebastiani/tutoriais/master/Dados/"
fileName   <- "Dados_Apply.R"

# load("Dados_Apply.Rdata")
source(paste0(urlRemote, pathGithub, fileName))

## Vetores
# Vetor numérico com apenas valores 1
VETOR1
[1] 1 1 1

# Vetor numérico com apenas valores 4
VETOR2
[1] 4 4 4

# Vetor numérico com diferentes valores
VETOR3
```

```
[1] 5 2 1

## Data.frames
# Data.frame com três variáveis numéricas, sendo duas fatores e uma numérica
DATAFRAME1
   F1 F2    V1
1   A  1 0.790
2   A  2 0.102
3   A  3 0.083
4   A  1 0.351
5   A  2 0.700
6   A  3 0.697
7   B  1 0.066
8   B  2 0.429
9   B  3 0.250
10  B  1 0.502
11  B  2 0.530
12  B  3 0.101

# Data.frame com três variáveis numéricas
DATAFRAME2
     V1    V2    V3
1 0.367 0.491 0.758
2 0.420 0.647 0.867
3 0.515 0.815 0.138
4 0.878 0.260 0.294
5 0.953 0.430 0.412

## Matrizes
# Matriz numérica completa
MATRIZ1
     Col1 Col2 Col3 Col4
Lin1 4.26 2.69 2.53 3.32
Lin2 2.22 1.45 2.23 4.05
Lin3 2.80 3.12 0.32 2.04
Lin4 3.26 2.33 2.01 2.03
Lin5 0.77 0.63 3.78 1.72

# Matriz numérica com NA
MATRIZ2
     Col1 Col2 Col3 Col4
Lin1    1    3    3    2
Lin2    5    2    2    4
Lin3    1    4    3    2
Lin4   NA    2    1    2
Lin5    4    1    2    1

## Listas
# Lista de comprimento três com vetores numéricos
LISTA1
$Lista1
 [1] 1 7 8 2 2 0 2 6 0 0
```

```
$Lista2
 [1] 4 7 3 3 2 4 4 5 6 1

$Lista3
 [1] 2 7 1 6 2 2 5 2 4 6

# Lista de comprimento três com matrizes numéricas
LISTA2
$Lista1
   Col1 Col2 Col3
L1 0.24 0.94 0.59
L2 0.89 0.69 0.53
L3 0.88 0.84 0.98
L4 0.81 0.38 0.20
L5 0.63 0.39 0.84

$Lista2
   Col1 Col2 Col3
L1 0.10 0.39 0.97
L2 0.38 0.36 0.56
L3 0.05 0.42 0.57
L4 0.15 0.32 0.64
L5 0.77 0.55 0.82

$Lista3
   Col1 Col2 Col3
L1 0.75 0.16 0.31
L2 0.43 0.14 0.71
L3 0.98 0.75 0.61
L4 0.56 0.35 0.38
L5 0.29 0.76 0.32
```

3 Os argumentos genéticos da família *Apply*

3.1 A argumentos FUN e ...

Nas família *Apply* um dos argumentos passados é sempre o nome de uma outra função, passada usando o argumento **FUN**, as funções dessa família então aplicam a função passada na estrura de dados fornecida. Qualqueis funções, do **R** ou definidas pelo próprio usuário, podem ser aplicadas à estrutura. O argumento **...** é usado para passar outros argumento da função passada no argumento **FUN**. Os operadores binários básicos como +, -, *, / e [também podem ser passados pelo argumento **FUN** desde que entre aspas duplas ("") ou simples (').

3.2 Argumento MARGIN

Função como **apply** e **sweep** aplicam funções nas margens de matrizes, então é preciso definir qual das margem da matriz será feita a aplicação. Se **MARGIN = 1** a função é aplicada nas linhas, se **MARGIN = 2** será aplicada nas colunas e se **MARGIN = c(1, 2)** será aplicada das linhas e colunas.

3.3 Argumento simplify

Algumas funções, como por exemplo **sapply**, **mapply** e **replicate**, simplificam os resultados da aplicação quando possível. Por exemplo, se o resultado de cada aplicação de uma função for um único valor os

resultados podem ser apresentados da forma de um único vetor, mantendo a posição de aplicação. Entretanto, caso o resultado de cada aplicação da função produzir sempre um vetor de mesmo comprimento, os resultados podem ser simplificados em uma matriz, retornando dessa maneira uma matriz $n\ x\ p$, onde n é o número de resultados e p o número de aplicações. Por fim, quando a aplicação retorna vetores de tamanhos desiguais, data.frames, matrizes ou listas, não há como simplificar os resultados então eles são retornados como listas. O argumento **simplify** determina o comportamento da simplificação, sendo que quando **simplify = FALSE** sempre uma lista é retornada.

3.4 Definições de funções

Normalmente as funções são atribuídas e salvas em objetos, são declaradas com a palavra **function**, os argumentos, obrigatórios ou opcionais, são definidos na declaração da função entre parênteses (**()**). Devido ao fato que normalemnte as funções possuem mais de uma linha de código, os blocos de código da função são especificados entre chaves (**{}**) e a função **return** especifica o que é retornado.

De maneira simplificada, as funções podem ser definidas minimamente. Essa definição mínima é viável apenas para funções com apenas uma linha de código, neste caso, não são usadas nem as chaves (**{}**) para especificar o bloco de códigos nem a função **return**, já que apenas o objeto resultante da única linhas de código é retornado. Ainda, as funções temporárias não precisam ser atribuidas em nenhum objeto.

```
# Atribuição de uma típica função
funcao.tipica <- function(x){
  result <- x+1
  return(result)
}

# Função mímina, apenas com os elementos essenciais
funcao.minima <- function(x) x+1

# Função temporária, apenas com os elementos essenciais
function(x) x+1
```

4 apply() - Aplicar funções nas margens de matrizes

A função **apply** aplica funções nas margens de matrizes. É preciso definir qual das margem da matriz a função será aplicada: se **MARGIN = 1** a função é aplicada nas linhas, se **MARGIN = 2** será aplicada nas colunas, ainda, se **MARGIN = c(1,2)** a função é aplicada das linhas e colunas. Além disso, é preciso difinir qual função será aplicada usando o argumento **FUN** e quaisquer outros argumentos da função a ser aplicada devem ser passados pelo argumento **...**.

```
apply(X, MARGIN, FUN, ...)
```

```
# Nesse exemplo são usados os objetos:
MATRIZ1
     Col1 Col2 Col3 Col4
Lin1 4.26 2.69 2.53 3.32
Lin2 2.22 1.45 2.23 4.05
Lin3 2.80 3.12 0.32 2.04
Lin4 3.26 2.33 2.01 2.03
Lin5 0.77 0.63 3.78 1.72
MATRIZ2
     Col1 Col2 Col3 Col4
Lin1    1    3    3    2
Lin2    5    2    2    4
```

```
Lin3    1    4    3    2
Lin4   NA    2    1    2
Lin5    4    1    2    1

# Aplicar função sd (desvio padrão) nas linhas (MARGIN = 1)
apply(MATRIZ1, MARGIN = 1, FUN = sd)
     Lin1      Lin2      Lin3      Lin4      Lin5
0.7846443 1.1038833 1.2515058 0.5868773 1.4530543

# Aplicar função sd nas colunas (MARGIN = 2)
apply(MATRIZ1, MARGIN = 2, FUN = sd)
    Col1     Col2     Col3     Col4
1.294728 1.001139 1.242308 1.003579

# Quando há argumentos extras eles podem ser especificado usando o argumento ...
# Na MATRIZ2 que há um NA, este que poderia ser removido do cálculo ...
apply(MATRIZ2, MARGIN = 2, FUN = sd)
    Col1     Col2     Col3     Col4
      NA 1.140175 0.836660 1.095445

# ... usando a opção na.rm = TRUE
# Note que na.rm é um argumento da função sd, passado pelo argumento ... da função apply
apply(MATRIZ2, MARGIN = 2, FUN = sd, na.rm = TRUE)
    Col1     Col2     Col3     Col4
2.061553 1.140175 0.836660 1.095445

# As funções também pode ser definidas pelo usúario
# Função para contar quantos NA há em um objeto
apply(MATRIZ2, MARGIN = 2, FUN = function(x) sum(is.na(x)))
Col1 Col2 Col3 Col4
   1    0    0    0

# Note que é preciso definir minimamente a função usando function
apply(MATRIZ2, 2, FUN = sum(is.na(x)))
Error in match.fun(FUN): object 'x' not found
```

5 tapply() - Dividir e aplicar funções a cada subconjunto

A função **tapply** divide as estruturas de dados e aplica funções a cada subconjunto, geralmente as funções passada calculam estatísticas descritivas. O argumento **INDEX** especifica um ou mais fatores para dividir os elementos da estrutura. O funcionamento dessa função é similar as funções **aggregate** e **by**.

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

```
# Nesse exemplo é usados o objeto:
DATAFRAME1
   F1 F2    V1
1   A  1 0.790
2   A  2 0.102
3   A  3 0.083
4   A  1 0.351
5   A  2 0.700
```

```
6   A  3 0.697
7   B  1 0.066
8   B  2 0.429
9   B  3 0.250
10  B  1 0.502
11  B  2 0.530
12  B  3 0.101

# Dividir o vetor V1 pelo fator F1 e calcular média
tapply(DATAFRAME1$V1, INDEX = DATAFRAME1$F1, FUN = mean)
        A         B
0.4538333 0.3130000

# Dividir o vetor V1 pelos fatores F1 e F2 e calcular média
tapply(DATAFRAME1$V1, INDEX = list(DATAFRAME1$F1, DATAFRAME1$F2), FUN = mean)
       1      2      3
A 0.5705 0.4010 0.3900
B 0.2840 0.4795 0.1755
```

6 sweep() - Percorrer uma matriz aplicando uma estatística em cada margem

A função **sweep** serve para percorrer uma matriz aplicando uma estatística em cada margem da matriz (linha ou coluna). O exemplo mais básico da sua utilidade é na padronização do tipo *z-score* de uma variável, essa padronização pode ser aplicada simultaneamente em todas as variáveis de uma matriz. Para padronizar, cada observação/célula deve ser subtraída da média da variável e então, dividida pelo desvio padrão da variável. Primeiramente é preciso definir a margem da aplicação (**MARGIN = 1** para linhas ou **MARGIN = 2** para colunas). O argumento **STATS** é um vetor que define a estatística que será aplicada, por padrão deve ter o mesmo comprimento que a margem da aplicação, entretanto o argumento **check.margin** pode alterar esse comportamento.

```
sweep(x, MARGIN, STATS, FUN, ...)
```

```
# Nesse exemplo é usados o objeto:
MATRIZ1
     Col1 Col2 Col3 Col4
Lin1 4.26 2.69 2.53 3.32
Lin2 2.22 1.45 2.23 4.05
Lin3 2.80 3.12 0.32 2.04
Lin4 3.26 2.33 2.01 2.03
Lin5 0.77 0.63 3.78 1.72

## Padronização z-score
# Calcular a média por colunas ...
res.media <- apply(MATRIZ1, MARGIN = 2, FUN = mean)
res.media
 Col1  Col2  Col3  Col4
2.662 2.044 2.174 2.632

# ... e desvio padrão usando apply
res.sd <- apply(MATRIZ1, MARGIN = 2, FUN = sd)
res.sd
```

```
    Col1     Col2     Col3     Col4
1.294728 1.001139 1.242308 1.003579

# Centralizar usando sweep...
res.centr <- sweep(MATRIZ1, MARGIN = 2, STATS = res.media, FUN = "-")
res.centr
       Col1   Col2   Col3   Col4
Lin1  1.598  0.646  0.356  0.688
Lin2 -0.442 -0.594  0.056  1.418
Lin3  0.138  1.076 -1.854 -0.592
Lin4  0.598  0.286 -0.164 -0.602
Lin5 -1.892 -1.414  1.606 -0.912

# ... então aplicar a divisão
res.centr.padro <- sweep(res.centr, MARGIN = 2, STATS = res.sd, FUN = "/")
res.centr.padro
           Col1       Col2        Col3       Col4
Lin1  1.2342363  0.6452648  0.28656332  0.6855467
Lin2 -0.3413845 -0.5933240  0.04507738  1.4129436
Lin3  0.1065861  1.0747755 -1.49238313 -0.5898890
Lin4  0.4618732  0.2856745 -0.13201232 -0.5998534
Lin5 -1.4613111 -1.4123908  1.29275475 -0.9087480
```

7 sapply() - Aplicar funções a cada elemento de um vetor

A função **sapply** aplica funções a cada elemento de um vetor, pode ser aplicada em vetores unidimensionais, data.frames (agrupamentos de vetores) e listas (também considerados vetores). Por padrão a função simplifica os resultados se possível, então pode retornar tanto único vetor, uma matriz ou listas. O argumento **simplify** determina o comportamento da simplificação, sendo que quando **simplify = FALSE** sempre é retornado uma lista.

```
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

```
# Nesse exemplo são usados os objetos:
VETOR1
[1] 1 1 1
VETOR2
[1] 4 4 4
VETOR3
[1] 5 2 1
DATAFRAME2
     V1    V2    V3
1 0.367 0.491 0.758
2 0.420 0.647 0.867
3 0.515 0.815 0.138
4 0.878 0.260 0.294
5 0.953 0.430 0.412
LISTA1
$Lista1
 [1] 1 7 8 2 2 0 2 6 0 0

$Lista2
 [1] 4 7 3 3 2 4 4 5 6 1
```

```
$Lista3
 [1] 2 7 1 6 2 2 5 2 4 6
LISTA2
$Lista1
   Col1 Col2 Col3
L1 0.24 0.94 0.59
L2 0.89 0.69 0.53
L3 0.88 0.84 0.98
L4 0.81 0.38 0.20
L5 0.63 0.39 0.84

$Lista2
   Col1 Col2 Col3
L1 0.10 0.39 0.97
L2 0.38 0.36 0.56
L3 0.05 0.42 0.57
L4 0.15 0.32 0.64
L5 0.77 0.55 0.82

$Lista3
   Col1 Col2 Col3
L1 0.75 0.16 0.31
L2 0.43 0.14 0.71
L3 0.98 0.75 0.61
L4 0.56 0.35 0.38
L5 0.29 0.76 0.32

# Aplicar a função log a cada elemento do vetor ...
sapply(VETOR3, FUN = log)
[1] 1.6094379 0.6931472 0.0000000

# ... outro exemplo. Note que length é igual a 1
sapply(VETOR3, FUN = length)
[1] 1 1 1

# Se resultado produzir apenas um valor, então pode ser simplicado para vetor ...
sapply(VETOR1, FUN = function(x) rnorm(x))
[1] -0.9633983  0.3764484 -0.9846738

# ... se produzir vetores de mesmos comprimento, então pode ser simplicado em matriz ...
# Note que matriz n x p, onde n é o número de resultados e p o número de aplicações
sapply(VETOR2, FUN = function(x) rnorm(x))
           [,1]       [,2]        [,3]
[1,]  0.8975594  0.4522813  1.92671883
[2,]  0.1292625 -0.6947379  0.05143036
[3,]  1.0337030 -0.2390136  1.80052376
[4,] -0.3422893 -1.0072990 -0.55145776

# ... se produzir vetores de comprimento diferentes não pode ser simplificado
sapply(VETOR3, FUN = function(x) rnorm(x))
[[1]]
[1]  0.1070589  0.7891240 -0.2442553  0.3841800 -1.1692355
```

```
[[2]]
[1] 1.51827243 0.07298118

[[3]]
[1] -0.527675

# Aplicar em data.frame (nas colunas)
sapply(DATAFRAME2, mean)
    V1     V2     V3
0.6266 0.5286 0.4938

# Aplicar em listas...
sapply(LISTA1, FUN = mean)
Lista1 Lista2 Lista3
   2.8    3.9    3.7

# .... para forçar a não simplificação dos resultados
sapply(LISTA1, FUN = mean, simplify = FALSE)
$Lista1
[1] 2.8

$Lista2
[1] 3.9

$Lista3
[1] 3.7

# Note que a função simplifica mesmo quando não seria conveniente ....
sapply(LISTA2, FUN = round, digits = 1)
      Lista1 Lista2 Lista3
 [1,]    0.2    0.1    0.8
 [2,]    0.9    0.4    0.4
 [3,]    0.9    0.0    1.0
 [4,]    0.8    0.1    0.6
 [5,]    0.6    0.8    0.3
 [6,]    0.9    0.4    0.2
 [7,]    0.7    0.4    0.1
 [8,]    0.8    0.4    0.8
 [9,]    0.4    0.3    0.3
[10,]    0.4    0.6    0.8
[11,]    0.6    1.0    0.3
[12,]    0.5    0.6    0.7
[13,]    1.0    0.6    0.6
[14,]    0.2    0.6    0.4
[15,]    0.8    0.8    0.3

# ... então é possível forçar a não simplificação
sapply(LISTA2, FUN = round, digits = 1, simplify = FALSE)
$Lista1
   Col1 Col2 Col3
L1  0.2  0.9  0.6
L2  0.9  0.7  0.5
L3  0.9  0.8  1.0
```

```
L4  0.8  0.4  0.2
L5  0.6  0.4  0.8

$Lista2
   Col1 Col2 Col3
L1  0.1  0.4  1.0
L2  0.4  0.4  0.6
L3  0.0  0.4  0.6
L4  0.1  0.3  0.6
L5  0.8  0.6  0.8

$Lista3
   Col1 Col2 Col3
L1  0.8  0.2  0.3
L2  0.4  0.1  0.7
L3  1.0  0.8  0.6
L4  0.6  0.3  0.4
L5  0.3  0.8  0.3
```

```
## Indexação com sapply

# Retornar linhas e coluna
sapply(LISTA2, FUN = "[", 2, 1)
Lista1 Lista2 Lista3
  0.89   0.38   0.43

# Retornar linha
sapply(LISTA2, FUN = "[", 2, )
     Lista1 Lista2 Lista3
Col1   0.89   0.38   0.43
Col2   0.69   0.36   0.14
Col3   0.53   0.56   0.71

# Retornar coluna
sapply(LISTA2, FUN = "[", , 3)
   Lista1 Lista2 Lista3
L1   0.59   0.97   0.31
L2   0.53   0.56   0.71
L3   0.98   0.57   0.61
L4   0.20   0.64   0.38
L5   0.84   0.82   0.32

# Retornar linha e coluna no formato de lista
sapply(LISTA2,"[", 2, 1, simplify = FALSE)
$Lista1
[1] 0.89

$Lista2
[1] 0.38

$Lista3
[1] 0.43
```

8 lapply() - Aplicar funções a cada elemento de um vetor, retornando lista

A função **lapply** funciona praticamente igual à **sapply** e permite a aplicação de uma função em cada elemento de um vetor unidimensional, data.frame ou lista. A principal diferença entre as elas é que a função **lapply** sempre retorna uma lista com os resultados. É preferível manter a função **sapply** na aplicação de vetores e **lapply** para aplicação de listas.

```
lapply(X, FUN, ...)
```

```
# Nesse exemplo são usados os objetos:
VETOR1
[1] 1 1 1
LISTA1
$Lista1
 [1] 1 7 8 2 2 0 2 6 0 0

$Lista2
 [1] 4 7 3 3 2 4 4 5 6 1

$Lista3
 [1] 2 7 1 6 2 2 5 2 4 6
LISTA2
$Lista1
   Col1 Col2 Col3
L1 0.24 0.94 0.59
L2 0.89 0.69 0.53
L3 0.88 0.84 0.98
L4 0.81 0.38 0.20
L5 0.63 0.39 0.84

$Lista2
   Col1 Col2 Col3
L1 0.10 0.39 0.97
L2 0.38 0.36 0.56
L3 0.05 0.42 0.57
L4 0.15 0.32 0.64
L5 0.77 0.55 0.82

$Lista3
   Col1 Col2 Col3
L1 0.75 0.16 0.31
L2 0.43 0.14 0.71
L3 0.98 0.75 0.61
L4 0.56 0.35 0.38
L5 0.29 0.76 0.32

# Aplicar a cada elemento da lista...
lapply(LISTA1, FUN = mean)
$Lista1
[1] 2.8

$Lista2
```

```
[1] 3.9

$Lista3
[1] 3.7

# ... outro exemplo
lapply(LISTA1, FUN = sum)
$Lista1
[1] 28

$Lista2
[1] 39

$Lista3
[1] 37

# Sempre retorna uma lista ...
lapply(VETOR1, FUN = function(x) rnorm(x))
[[1]]
[1] -0.5245256

[[2]]
[1] 0.7469007

[[3]]
[1] -1.182414

# ... outro exemplo com lista
lapply(LISTA2, FUN = round, digits = 1)
$Lista1
   Col1 Col2 Col3
L1  0.2  0.9  0.6
L2  0.9  0.7  0.5
L3  0.9  0.8  1.0
L4  0.8  0.4  0.2
L5  0.6  0.4  0.8

$Lista2
   Col1 Col2 Col3
L1  0.1  0.4  1.0
L2  0.4  0.4  0.6
L3  0.0  0.4  0.6
L4  0.1  0.3  0.6
L5  0.8  0.6  0.8

$Lista3
   Col1 Col2 Col3
L1  0.8  0.2  0.3
L2  0.4  0.1  0.7
L3  1.0  0.8  0.6
L4  0.6  0.3  0.4
L5  0.3  0.8  0.3
```

```
# Indexação com lapply

# Retornar linhas e coluna
lapply(LISTA2, FUN = "[", 2, 1)
$Lista1
[1] 0.89

$Lista2
[1] 0.38

$Lista3
[1] 0.43

# Retornar linha
lapply(LISTA2, FUN = "[", 2, )
$Lista1
Col1 Col2 Col3
0.89 0.69 0.53

$Lista2
Col1 Col2 Col3
0.38 0.36 0.56

$Lista3
Col1 Col2 Col3
0.43 0.14 0.71

# Retornar coluna
lapply(LISTA2, FUN = "[", , 3)
$Lista1
  L1   L2   L3   L4   L5
0.59 0.53 0.98 0.20 0.84

$Lista2
  L1   L2   L3   L4   L5
0.97 0.56 0.57 0.64 0.82

$Lista3
  L1   L2   L3   L4   L5
0.31 0.71 0.61 0.38 0.32

# Retornar linha e coluna em formato de vetor
unlist(lapply(LISTA2,"[", 2, 1 ))
Lista1 Lista2 Lista3
  0.89   0.38   0.43
```

9 mapply() - Aplicar funções a um ou mais vetores

A função **mapply** é a versão multivariada da **sapply**, nela são fornecidos um ou mais vetores. A função então é aplicada em cada um dos vetores fornecidos de maneira simultânea. Por exemplo, quando dois vetores são fornecidos a função é aplicada no primeiro elemento de ambos os vetores, após no segundo elemento de ambos os vetores e assim por diante. Nesta função, diferente das demais, os vetores são passados pelo argumento `...` e demais argumentos para a função aplicada são então passados usando o argumento

MoreArgs, ainda, o argumento **simplify** determina o comportamento da simplificação dos resultados.

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

```
# Nesse exemplo são usados os objetos:
VETOR3
[1] 5 2 1

# A função rep requer x e times ...
rep(x = 2, times = 5)
[1] 2 2 2 2 2

# ... então mapply permite passar ambos argumentos simultaneamente
mapply(FUN = rep, x = VETOR3, times = VETOR3)
[[1]]
[1] 5 5 5 5 5

[[2]]
[1] 2 2

[[3]]
[1] 1

# Outro exemplo
mapply(FUN = "+", VETOR3, VETOR3)
[1] 10  4  2

# O argumento MoreArgs permite passar outros argumentos de maneira fixa
mapply(rep, times = VETOR3, MoreArgs = list(x = 12))
[[1]]
[1] 12 12 12 12 12

[[2]]
[1] 12 12

[[3]]
[1] 12
```

10 replicate() - Replicar uma expressão

A função **replicate** permite a aplicação repetida de uma expressão. Seu uso é voltado para funções que retornam valores diferentes cada vez que são executadas, geralmente envolvendo geradores de números aleatórios. Seu uso é uma alternativa a loop/laço que são apenas repetições de uma mesma expressão com mesmos parâmetros. O argumento **expr** deve ser uma expressão completa para ser repetida, não apenas um função. O argumento **simplify** determina o comportamento da simplificação.

```
replicate(n, expr, simplify = "array")
```

```
# Função para lançar x moedas
f.moeda <- function(lances = 0){
  res <- sample(x= c("Cara", "Coroa"), size = lances, replace = TRUE)
  return(res)
}
```

```r
# Teste de f.moeda com 1 lançamento...
f.moeda(lances = 1)
[1] "Coroa"
# ... e 3 lançamentos
f.moeda(lances = 3)
[1] "Coroa" "Coroa" "Coroa"

# Replicar n vezes a expressão com 1 lançamento cada
# Note que retorna um vetor
replicate(n = 5, expr = f.moeda(lances = 1))
[1] "Coroa" "Coroa" "Cara"  "Coroa" "Coroa"

# Replicar n vezes a expressão com 8 lançamentos cada
# Note que matriz n x p, onde n é o número de resultados e p o número de replicações
replicate(n = 5, expr = f.moeda(lances = 8))
     [,1]    [,2]    [,3]    [,4]    [,5]
[1,] "Cara"  "Cara"  "Cara"  "Coroa" "Coroa"
[2,] "Cara"  "Coroa" "Cara"  "Cara"  "Cara"
[3,] "Cara"  "Cara"  "Coroa" "Coroa" "Cara"
[4,] "Coroa" "Cara"  "Cara"  "Coroa" "Cara"
[5,] "Cara"  "Cara"  "Coroa" "Coroa" "Cara"
[6,] "Coroa" "Cara"  "Coroa" "Cara"  "Coroa"
[7,] "Coroa" "Cara"  "Coroa" "Coroa" "Coroa"
[8,] "Coroa" "Coroa" "Cara"  "Cara"  "Cara"

# Para retornar uma lista simplify deve ser FALSE
replicate(n = 5, expr = f.moeda(lances = 8), simplify = FALSE)
[[1]]
[1] "Coroa" "Cara"  "Cara"  "Cara"  "Cara"  "Cara"  "Coroa" "Cara"

[[2]]
[1] "Coroa" "Cara"  "Cara"  "Coroa" "Cara"  "Cara"  "Coroa" "Cara"

[[3]]
[1] "Coroa" "Coroa" "Cara"  "Cara"  "Cara"  "Cara"  "Cara"  "Cara"

[[4]]
[1] "Cara"  "Cara"  "Coroa" "Coroa" "Coroa" "Coroa" "Coroa" "Cara"

[[5]]
[1] "Cara"  "Cara"  "Coroa" "Cara"  "Coroa" "Cara"  "Coroa" "Cara"

# Diferentemente das demais funções, expr não pode ser uma função...
# ... caso for, replica a própria função ...
replicate(n = 2, expr = f.moeda)
[[1]]
function(lances = 0){
  res <- sample(x= c("Cara", "Coroa"), size = lances, replace = TRUE)
  return(res)
}
<bytecode: 0x7ff537189238>

[[2]]
```

```
function(lances = 0){
  res <- sample(x= c("Cara", "Coroa"), size = lances, replace = TRUE)
  return(res)
}
<bytecode: 0x7ff537189238>

# ... ou retorna um erro
replicate(n = 2, expr = f.moeda, lances = 8)
Error in replicate(n = 2, expr = f.moeda, lances = 8): unused argument (lances = 8)
```

11 Outras funções da família *Apply*

Outras funções, da família ou assiciadas, também compartilham a mesma lógica e podem ser aplicadas de maneira similar, como por exemplo: **aggregate**, **by**, **vapply**, **rapply**, **Map**, **Reduce**, **outer**, entre outras.

12 Teste de desempenho

O desempenho das funções, ou qualquer código, podem ser avaliadas usando *benchmark*, neste o desempenho é avaliado de maneira relativa, execuranto uma série de testes. O pacote *microbenchmark* avalia o desemprenho de funções no R, retornando o tempo em microssegundos ou nanossegundos que a série de testes leva para ser executada. Abaixo são mostrado os resultados de testes simples, comparando o desempenho relativo de funções que utilizam os loop/laços definidos explicitamente usando *for* com funções da família *Apply*.

```
require(microbenchmark)

# Função que utiliza loop em vetores
# Note que o vetor de resultados é inteiramente definido antes do loop
f.loop.vetor <- function(V){
  nv <- length(V)
  res <- vector("numeric", nv)
  res[] <- NA
  for(i in 1:nv){
    res[i] <- V[i]+1
  }
  return(res)
}

# Função que utiliza loop em vetores
# Note que o vetor de resultados não é inteiramente definido antes do loop...
# ... mas concatenado em cada etapa do loop
f.loop.vetor.concatenar <- function(V){
  nv <- length(V)
  res <- c()
  for(i in 1:nv){
    res <- c(res, V[i]+1)
  }
  return(res)
}

# Função que utiliza sapply em vetores
```

```r
f.apply.vetor <- function(V){
  res <- sapply(V, function(x) x+1)
  return(res)
}

# Função que utiliza um loop aninhado em uma matriz
f.loop.matriz <- function(M){
  nr <- nrow(M)
  nc <- ncol(M)
  RES <- matrix(NA, nr, nc)
  for (i in 1:nr) {
    for (j in 1:nc) {
      RES[i,j] <- M[i,j]+1
    }
  }
  return(RES)
}

# Função que utiliza sapply aninhado em uma matriz
f.apply.matriz <- function(M){
  nr <- nrow(M)
  nc <- ncol(M)
  RES <- sapply(1:nc, function(j) sapply(1:nr, function(i) M[i,j]+1))
  return(RES)
}

# Nos resultados abaixo note principalmente o tempo médio da aplicação de cada função
# Cada teste foi conduzido 100 vezes, o tempo é mostradoem milissegundo (ms)

# Teste 1 - Vetor de tamanho 100
VETOR.TESTE1 <- rgamma(100, shape = 2)
microbenchmark(loop = f.loop.vetor(VETOR.TESTE1),
               loop.con = f.loop.vetor.concatenar(VETOR.TESTE1),
               apply = f.apply.vetor(VETOR.TESTE1),
               times = 100, unit = "ms")
Unit: milliseconds
     expr      min       lq       mean    median        uq      max neval cld
     loop 0.007307 0.0082785 0.05974529 0.0088000 0.0096980 5.036934   100   a
 loop.con 0.043321 0.0472940 0.11446897 0.0550125 0.0625755 5.409167   100   a
    apply 0.067207 0.0711285 0.10691543 0.0728810 0.0757830 3.267355   100   a

# Teste 2 - Vetor de tamanho 10000
VETOR.TESTE2 <- rgamma(10000, shape = 2)
microbenchmark(loop = f.loop.vetor(VETOR.TESTE2),
               loop.con = f.loop.vetor.concatenar(VETOR.TESTE2),
               apply = f.apply.vetor(VETOR.TESTE2),
               times = 100, unit = "ms")
Unit: milliseconds
     expr        min         lq        mean      median         uq       max neval cld
     loop   0.580782   0.635273   0.6860848   0.6534985   0.696536   1.23734   100  a
 loop.con 360.526141 369.216811 413.1174941 381.1589260 420.151806 690.80105   100   b
    apply   5.766909   5.955110   6.4942667   6.0971075   6.311769  19.72322   100  a
```

```
# Teste 3 - Matriz de tamanho 10 x 10
MATRIZ.TESTE1 <- matrix(rgamma(10, 2), 10, 10)
microbenchmark(loop = f.loop.matriz(MATRIZ.TESTE1),
               apply = f.apply.matriz(MATRIZ.TESTE1),
               times = 100, unit = "ms")
Unit: milliseconds
  expr      min        lq       mean    median       uq      max neval cld
  loop 0.016836 0.0182705 0.08793295 0.0193185 0.020891 6.700178   100  a
 apply 0.194434 0.1978645 0.29281822 0.2067905 0.220488 7.304718   100   b

# Teste 4 - Matriz de tamanho 1000 x 1000
MATRIZ.TESTE2 <- matrix(rgamma(10000, 2), 1000, 1000)
microbenchmark(loop = f.loop.matriz(MATRIZ.TESTE2),
               apply = f.apply.matriz(MATRIZ.TESTE2),
               times = 100, unit = "ms")
Unit: milliseconds
  expr       min       lq     mean   median       uq       max neval cld
  loop  97.17957 108.5327 133.7112 114.9223 130.5094  506.1534   100  a
 apply 743.87571 803.8995 927.9384 877.6006 944.1704 2074.6906   100   b
```

O desempenho das funções depende de vários aspectos, variando incluvide dentro das diferentes funções da família *Apply*. Nos testes as funções que usam *for* para definir os loop/laços de maneira explícita apresentam desempenho superior em relação as da família *Apply*. Note que o uso do da função para concatenar (**c**) dentro do loop piora o desempenho consideravelmente.

13 Guia de ajuda rápida

```
# Família Apply
apply() - Aplicar funções nas margens de matrizes
tapply() - Dividir e aplicar funções a cada subconjunto
sweep() - Percorrer uma matriz aplicando uma estatística em cada margem
sapply() - Aplicar funções a cada elemento de um vetor
lapply() - Aplicar funções a cada elemento de um vetor, retornando lista
mapply() - Aplicar funções a um ou mais vetores
replicate() - Replicar uma expressão

# Benchmark
microbenchmark() - Realizar teste de benchmark em códigos ou funções
```

14 Conclusão

O objetivo deste texto foi apenas apresentar as funções básicas da família *Apply* e comparar o desenhenho delas usando o pacote *microbenchmark*. O desempenho depende da aplicação, mas em geral os loop/laços declarados de maneira explícita apresentam desempenho superior. Por outro lado, as funções *Apply* são alternativas úteis para muitos problemas, são fácies de executar e necessitam de poucas linhas de código. Espero que este texto tenha sido útil e, por favor, avise-me se tiver dúvidas ou sugestões sobre este texto.

15 Mais informações

Outros textos e tutoriais sobre R podem ser encontrados em https://vanderleidebastiani.github.io/tutoriais.

16 Referências

- Crawley, Michael J. 2007. **The R book**. John Wiley & Sons, Chichester.
- R Core Team; 2018. **R Language Definition**. https://cran.r-project.org/doc/manuals/R-lang.html
- Mersmann, Olaf.; 2019. **microbenchmark: Accurate Timing Functions.**